Dev Environments & Python Applications

One of the goals of this class is to start moving you further beyond the theoretical and into the real world of how we use code to achieve our goals.

To that end we're starting in a very practical place of how to go from scripts to applications.

We need to first take a step back and talk about the environment in which programs are being run.

1. Development Environments

Development Environment

A machine with all of the tools we need to write, build, and test software.

Remote Development Environments

University Linux Servers
Python Anywhere
GitHub Codespaces
Google CoLab

In 2024 there are a lot of great ways to not need to set up a local development environment when you're first learning to program.

You can `ssh` into CS department servers and use their pre-configured environment. Most editors, including VSCode, make this easy with remote editing plugins that negate the need to copy files back and forth. You can use the environment provided to you, which is particularly helpful for debugging because your environment will be (nearly) identical to your peers.

Of course, there are also tools like Python Anywhere and Google's Colab Notebooks that allow editing and running Python code in the browser, and running it on remote servers. These tools are great for beginners, allowing them to focus on coding, or actually solving their problem. They remain useful for professionals as well, particularly when it comes to sharing work publicly, but there are a number of reasons that nearly all developers spend most of their time working in a local development environment.

Remote

- Access from multiple devices
- Configuration can be someone else's problem
- Uniformity across team/institution

Local

- Typically faster
- Full control of tools
- Ability (but also need) to debug your own problems

Remote

Easier early on, when you are working with one or two files, but as projects grow, hit pain points that are difficult to solve.

Local

Requires a deeper understanding of what is going on that takes time but ultimately is needed either way.

Once up and running often easier than dealing with remote.

Why is this worth it?

- Testing & Debugging
- Better Understanding of System
- Automation

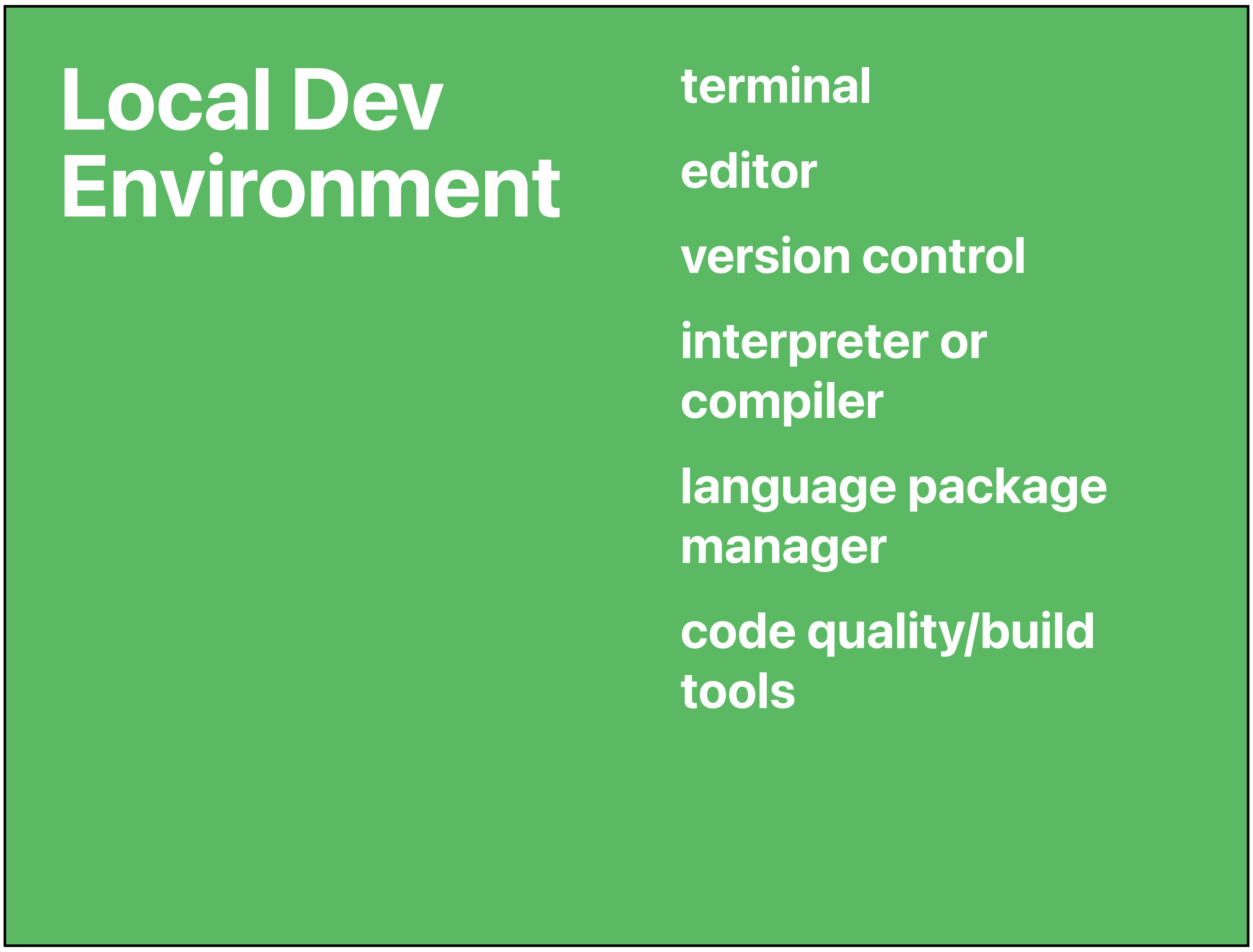
We spend a lot of our lives as programmers debugging things. The most critical skill a new programmer learns is how to come up with a reasonable hypothesis about why their code isn’t working and then take steps to prove or disprove that hypothesis.

When you are first learning, machines seem finicky and opaque, but at some point the veil of mystery must come off. A mechanic cannot avoid learning what is under the hood of a car any more than a programmer can avoid understanding their environment and tools.

Your personal machine is your workshop. To become proficient at the skill of programming, comfort with your tools is essential. Speed and proficiency are essential.

Remember, Neither Is Immune To Data Loss

commit **and push** early and often



Local Dev Environment

terminal

editor

version control

interpreter or compiler

language package manager

code quality/build tools

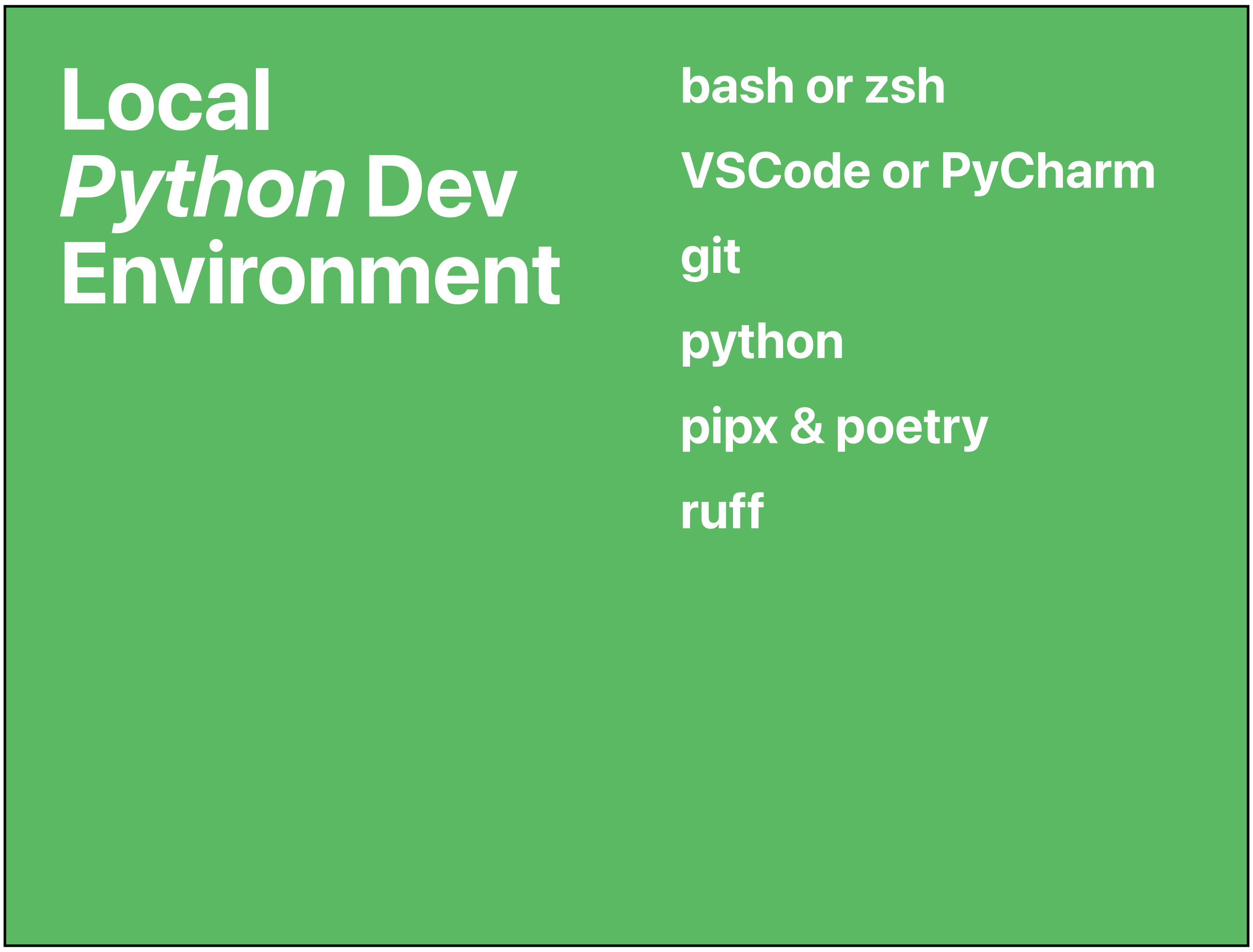
For our purposes here, I'm going to define this as having a physical machine that can run the code, not just edit it. For example, if you are writing an application that requires Python 3.11 and Django 5.0, the machine that has the Python 3.11 executable and Django 5.0 source installed on it.

Typically this would be your laptop. (Though this could just as easily include a server under your control.)

To set up a local Python development environment you'll need to ensure you have a few key things:

- a terminal
- version control
- a Python interpreter **under your control**
- a Python package manager
- any additional software build/code quality tools you need

Though the examples I'll be using are for Python, working in another language typically requires a very similar suite of tools. (For instance, python/poetry/ruff can be swapped out for nodejs/npm/prettier and you'd have a JS environment too.)

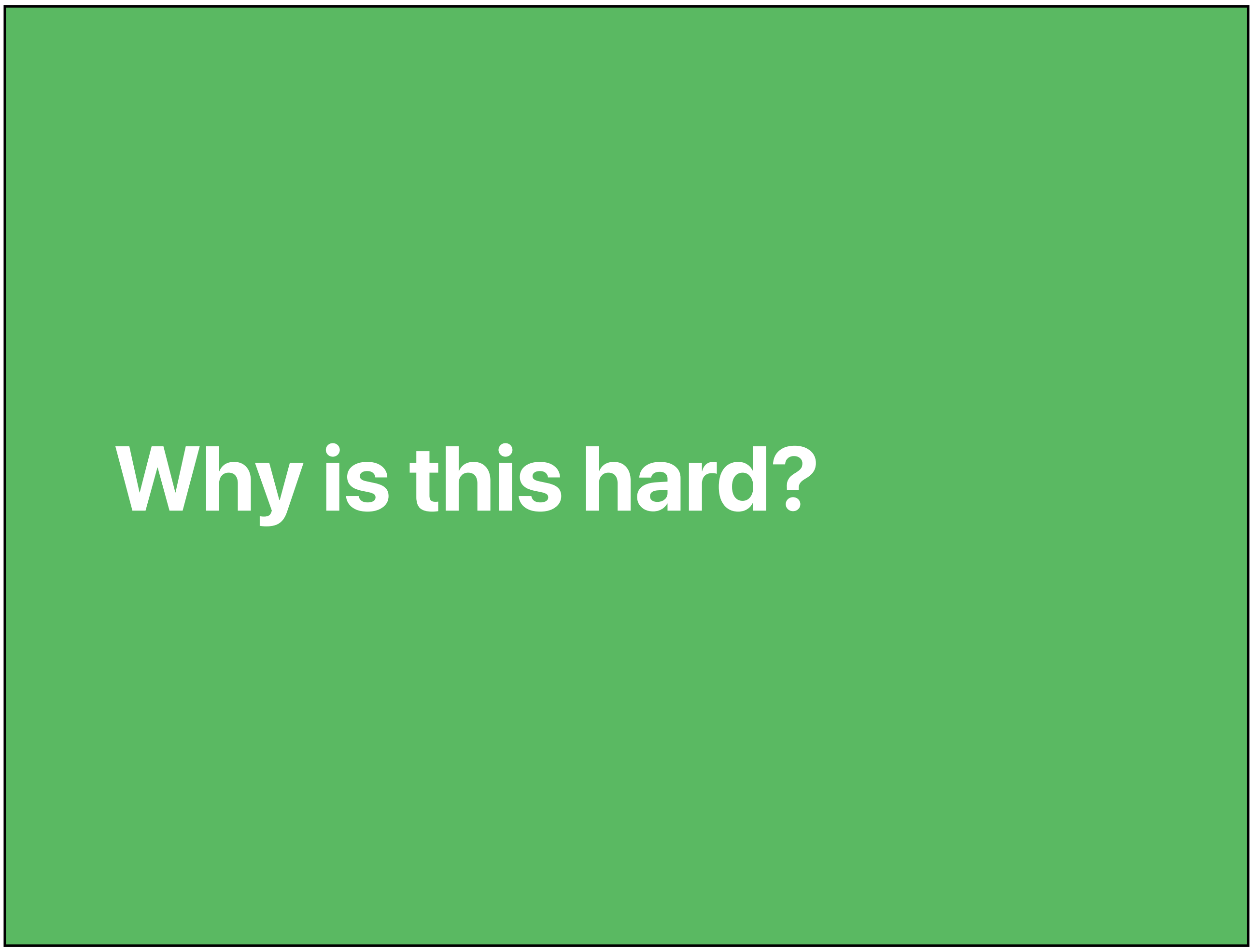


Local *Python* Dev Environment

- bash or zsh
- VSCode or PyCharm
- git
- python
- pipx & poetry
- ruff

These are the ones I am going to recommend for this course.

There are other options too which we'll touch on briefly.



Why is this hard?

Variety: The first reason is that everyone's machines vary a bit, and so giving general instructions that work for everyone is impossible.

Despite this, typically guides give a single set of commands, often without any explanation, and when a command goes wrong the student following the tutorial has no way to understand how to possibly fix it.

Lack of understanding of internals.

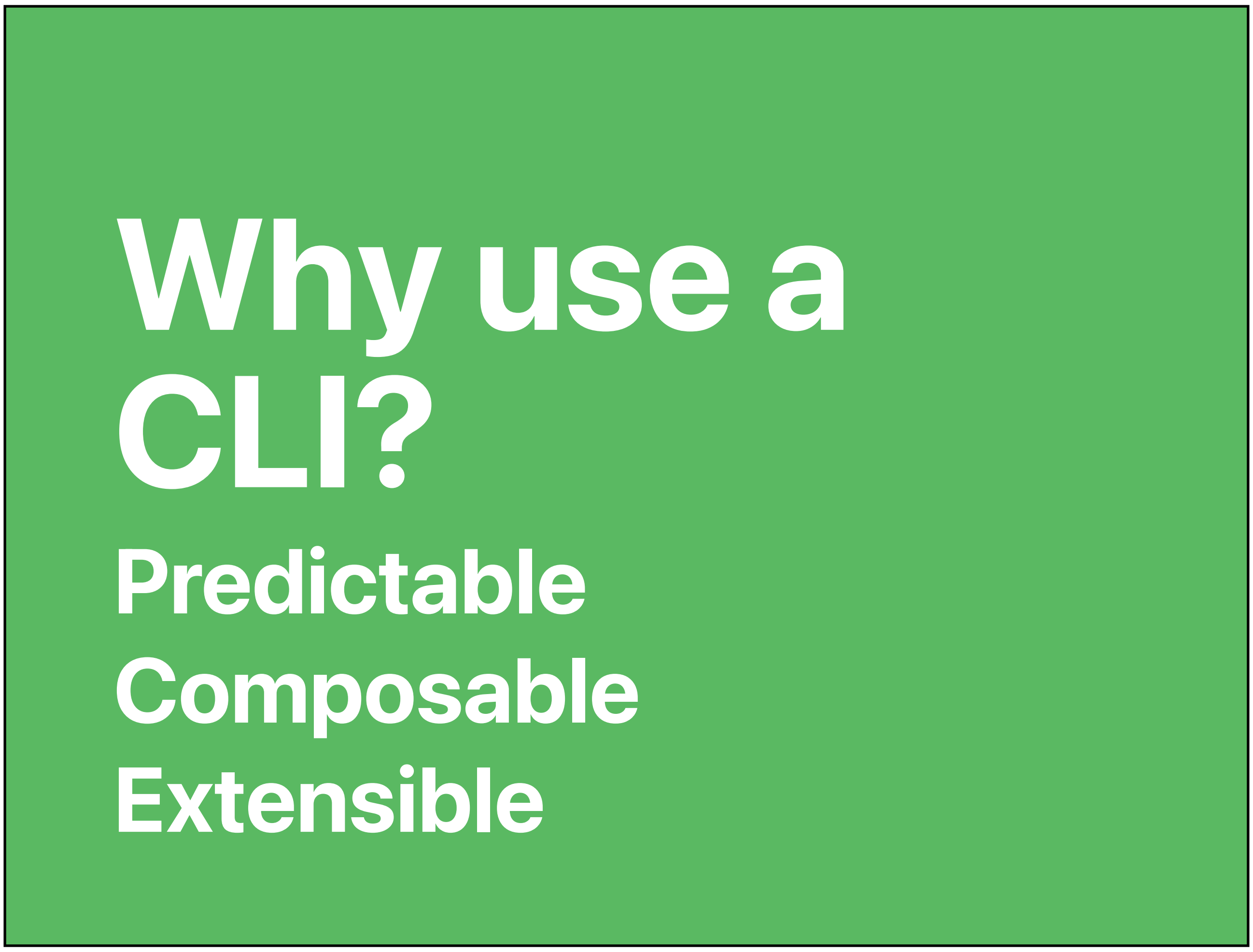
Because of this, I am going to attempt to explain the key concepts from the bottom up. That way if you get lost you hopefully have the tools to know where to begin.

2. CLI Revisited

Command Line Interfaces

Let's tackle the issue of not understanding some of the key concepts by revisiting the command line interface.

How many currently feel comfortable with it?



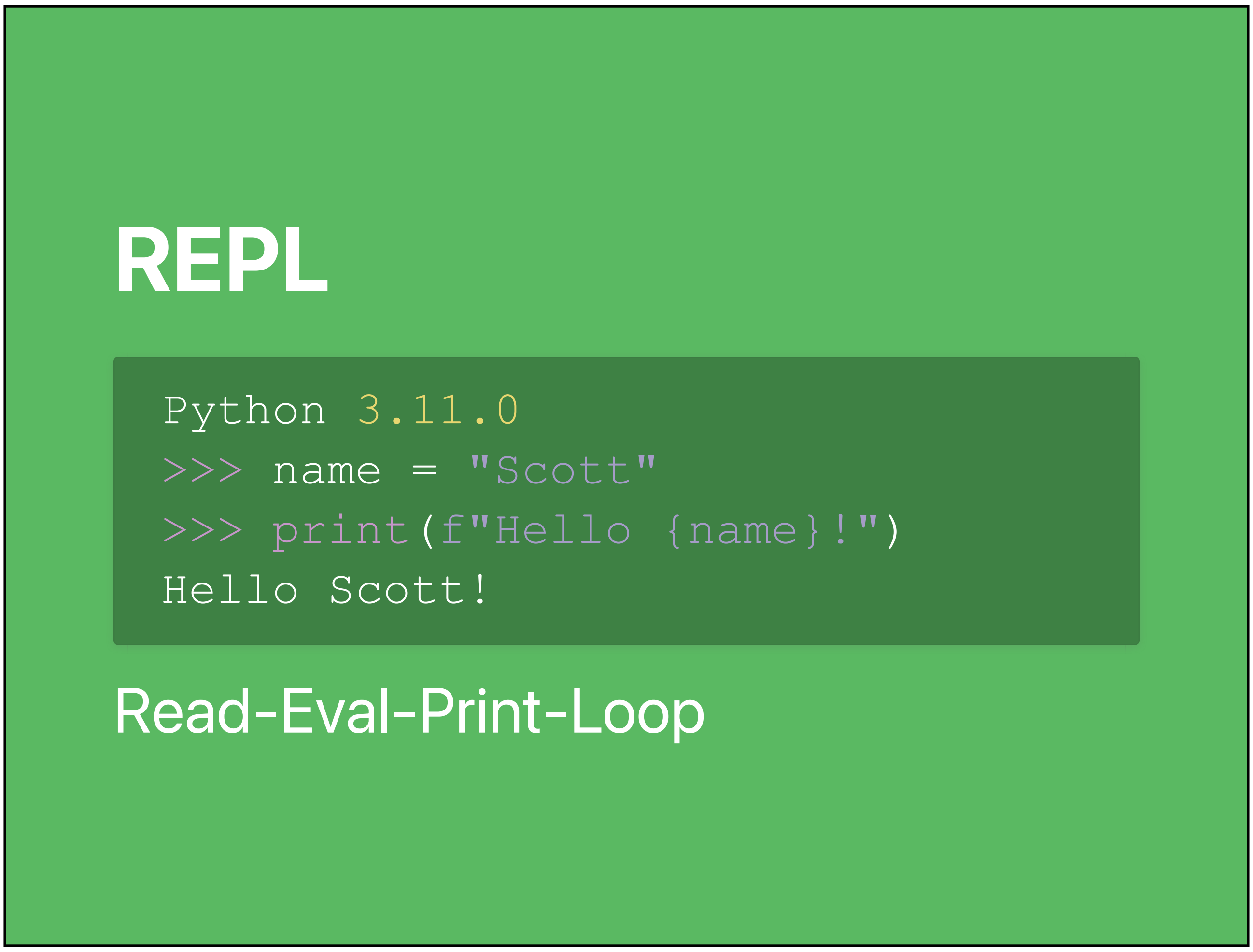
Why use a CLI?

Predictable
Composable
Extensible

Predictable: we can share them with one another in a way that would take a short video tutorial. They are less subject to UI-drift across versions.

Composable: we can modify/alter them. If we know how to crop a single image with the command line, we can automate this to call it hundreds of times to crop all of our photos.

Extensible: Easy to write our own commands.

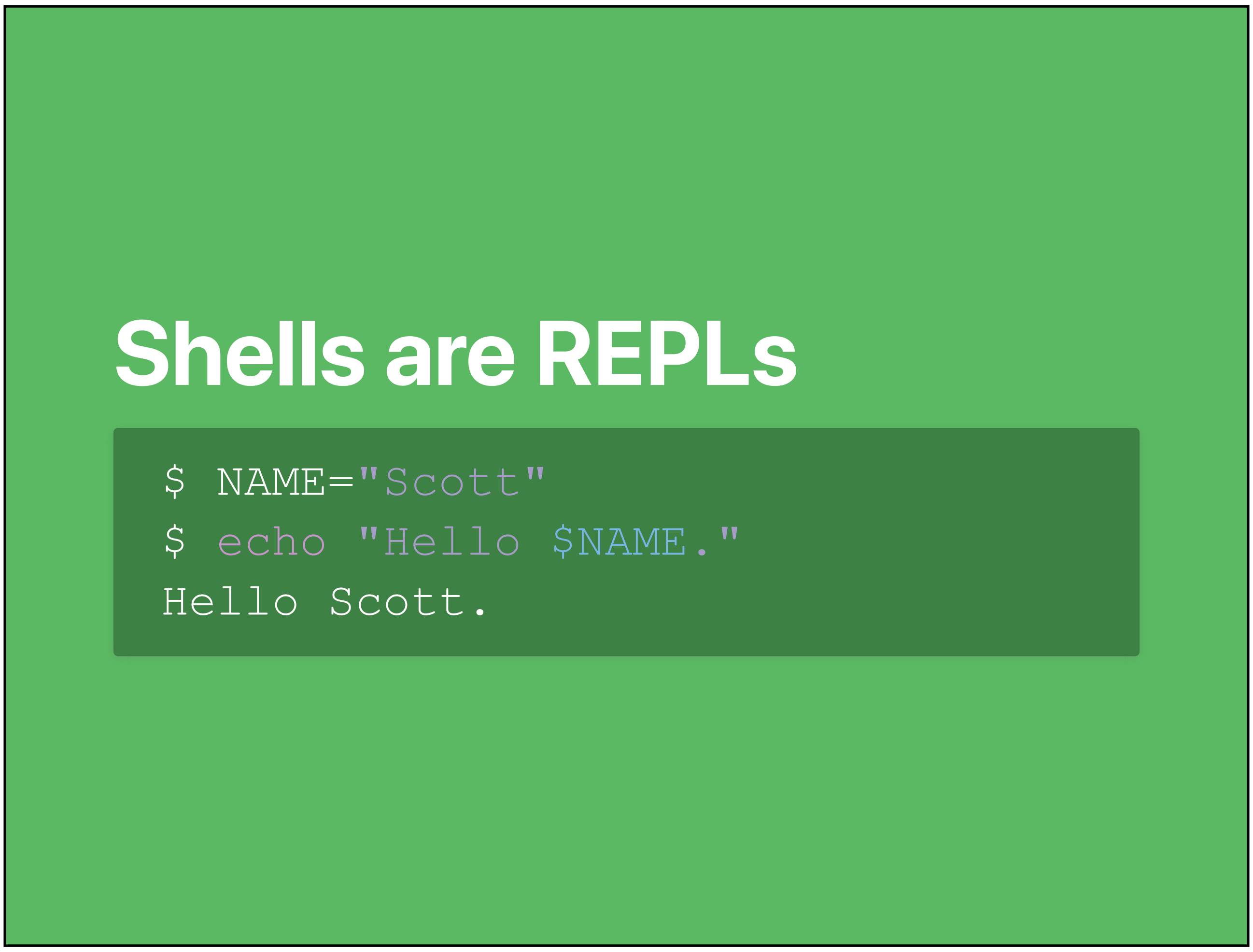


REPL

```
Python 3.11.0
>>> name = "Scott"
>>> print(f"Hello {name}!")
Hello Scott!
```

Read-Eval-Print-Loop

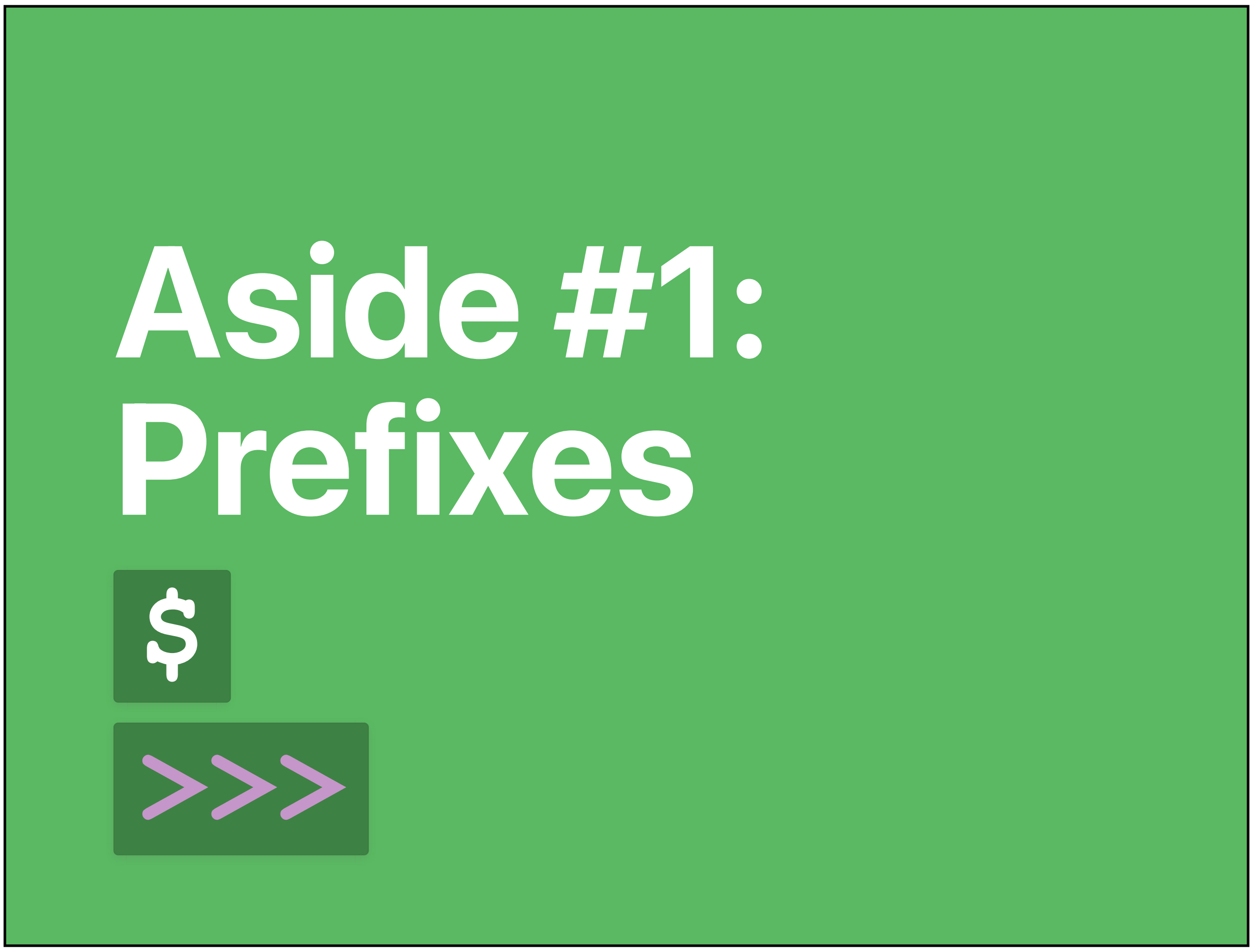
Refresher on what a REPL is.



Shells are REPLs

```
$ NAME="Scott"
$ echo "Hello $NAME."
Hello Scott.
```

Shells are REPLs too...



Aside #1: Prefixes

$

>>>

Quick aside to make sure we're all on the same page with code examples.

$ = sh

| | | = python
dollar sign vs. arrows

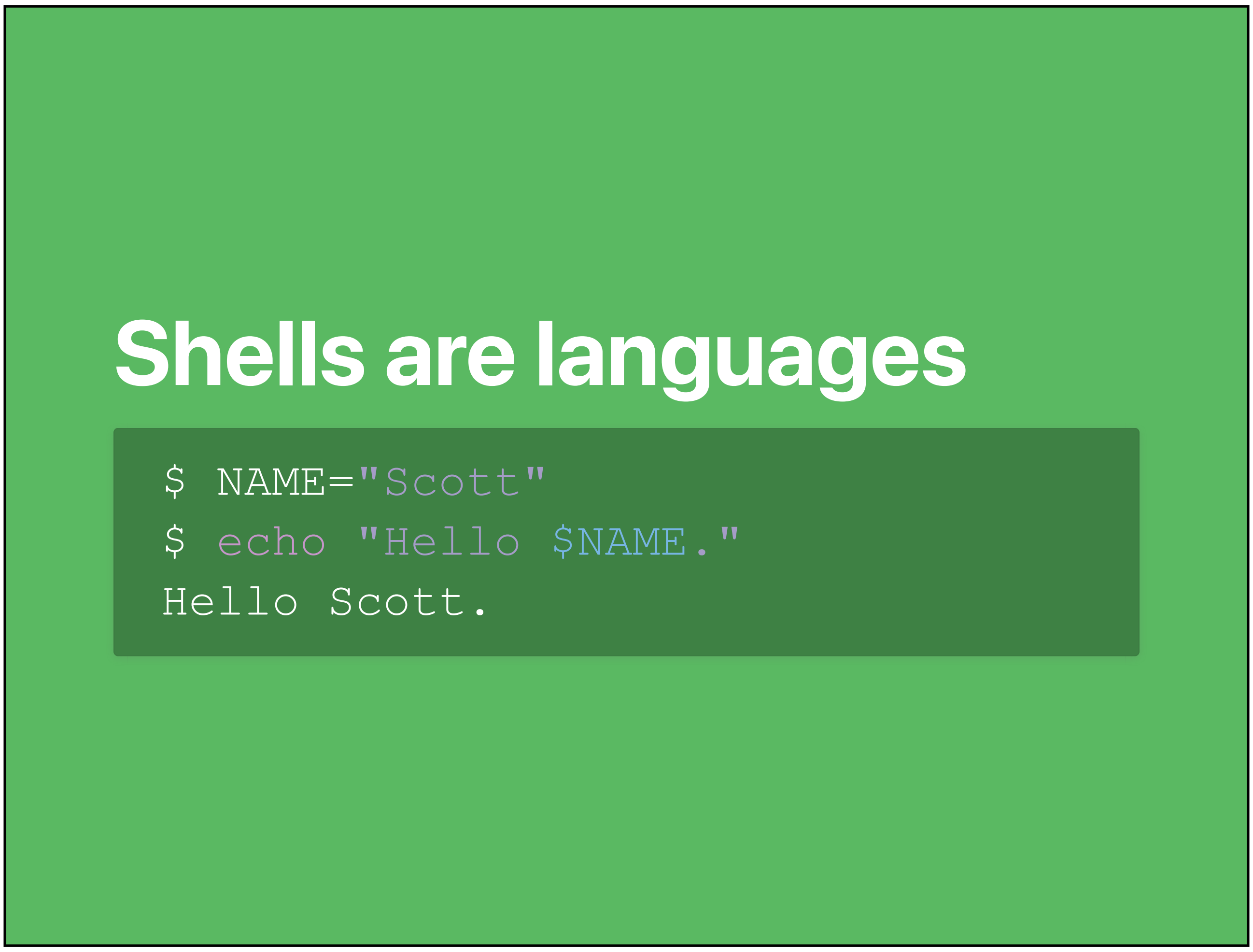
Aside #2: Windows*

Highly recommend using WSL on windows so you can use a POSIX shell (bash, zsh).

Powershell is a completely different language, and while it has some interesting features, 99% of instructions online will work in a POSIX environment.

Everything is just a little bit harder on Windows. But once you overcome the initial obstacles and have a functional environment, you should be able to use the other commands.

Not so if you stick to powershell.



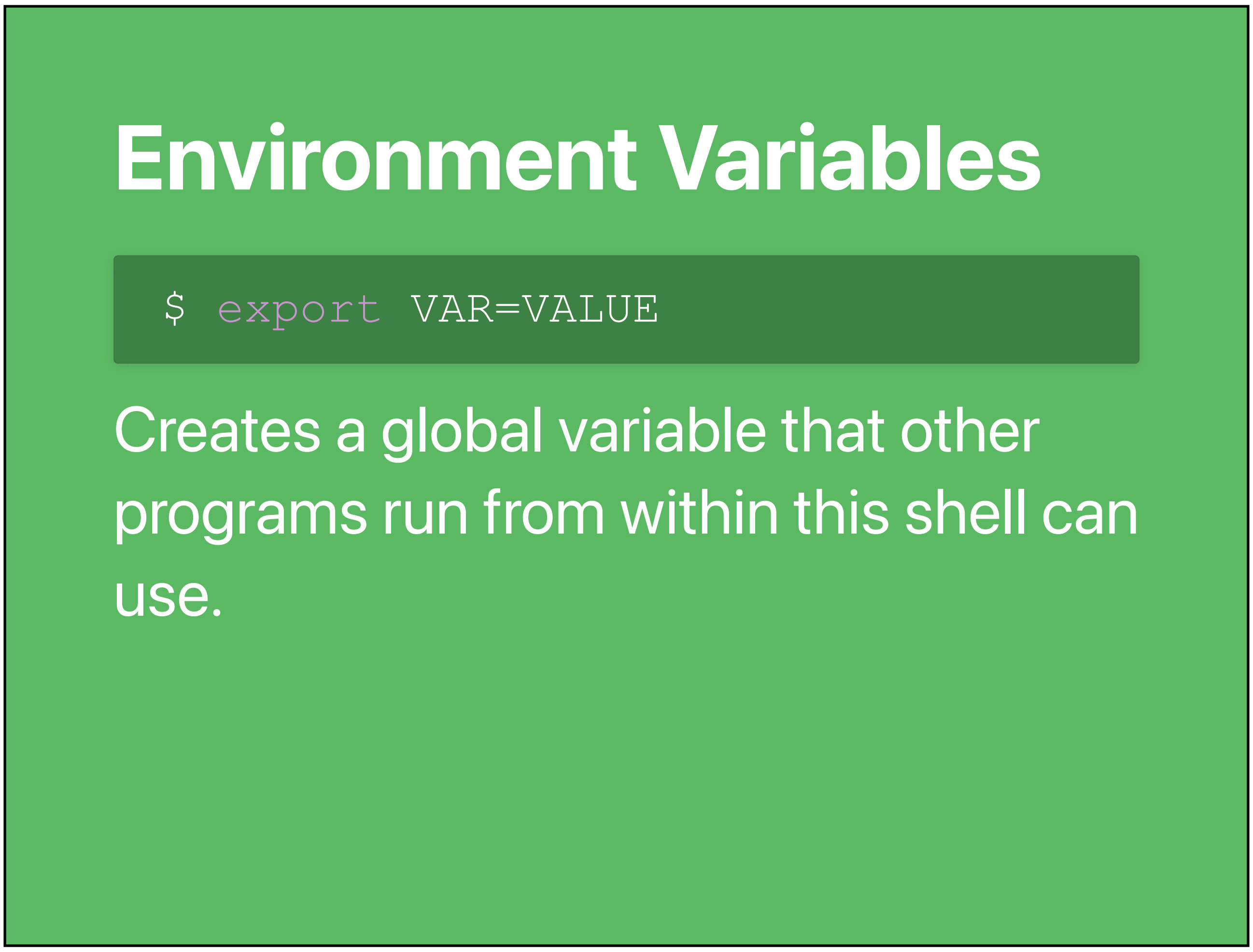
Shells are languages

```
$ NAME="Scott"
$ echo "Hello $NAME."
Hello Scott.
```

So to reemphasize:

Shells are REPLs…for another programming language.

That language has variables and function-ish things.



Environment Variables

```
$ export VAR=VALUE
```

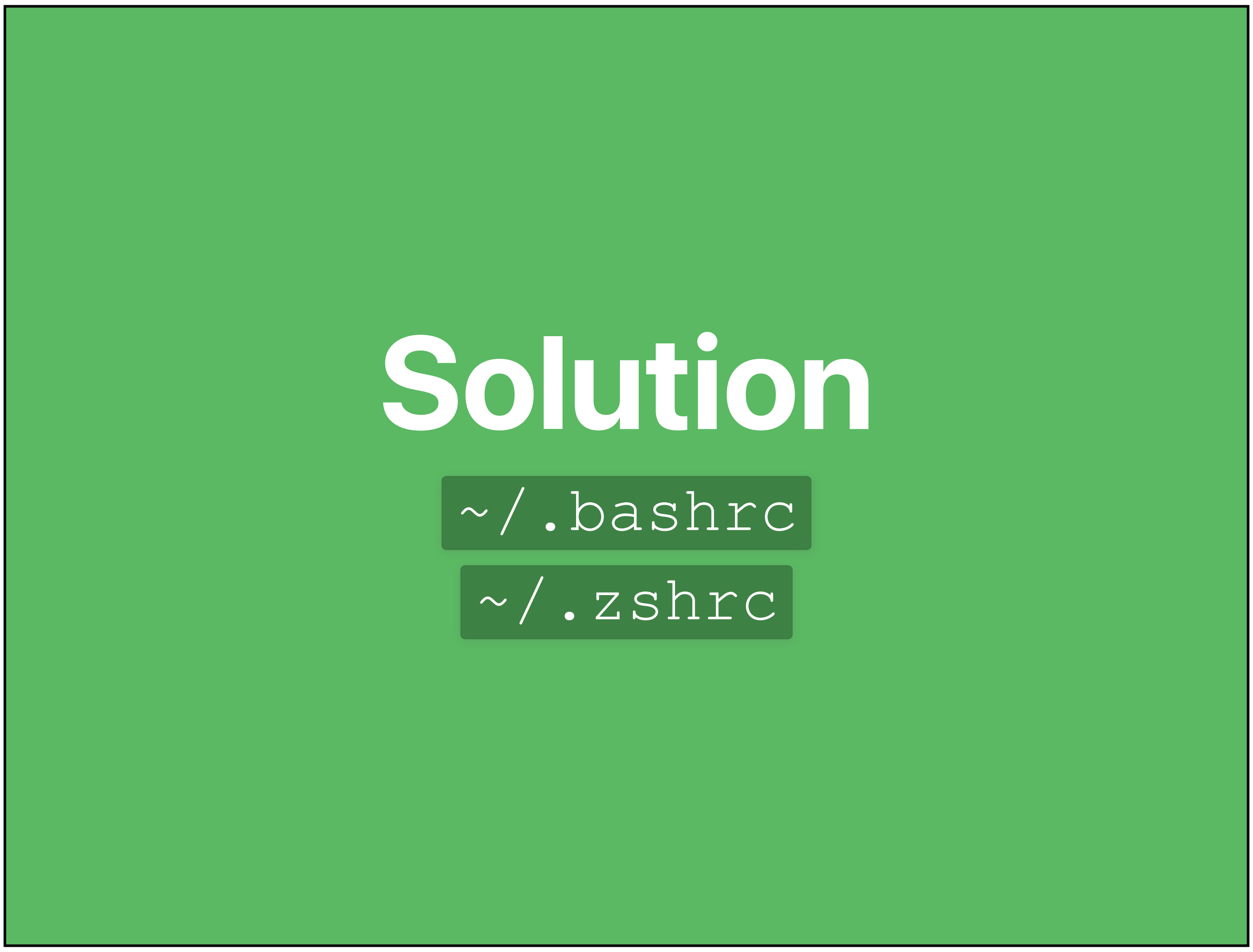
Creates a global variable that other programs run from within this shell can use.

A lot of instructions are going to assume you know how to set one of these.

Sessions

Every time you open a new terminal, you are in a new session.

```
$ export SOMEVAR=123
$ echo $SOMEVAR
123

# close terminal & re-open
$ echo $SOMEVAR
```

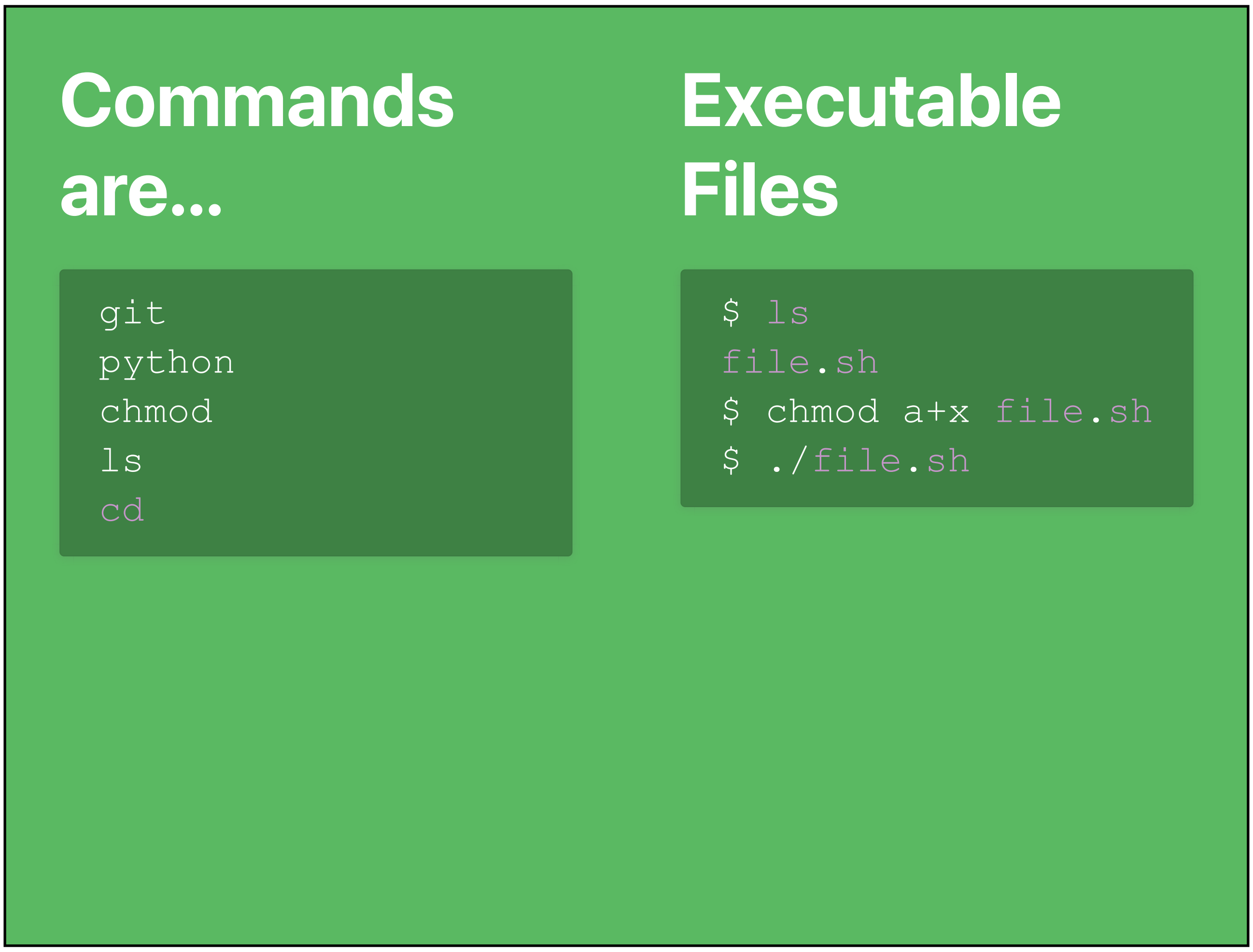


Solution

`~/.bashrc`

`~/.zshrc`

These files run every time you start a new terminal. You can set variables in those and they will be set in every new terminal.



Commands are...

```
git
python
chmod
ls
cd
```

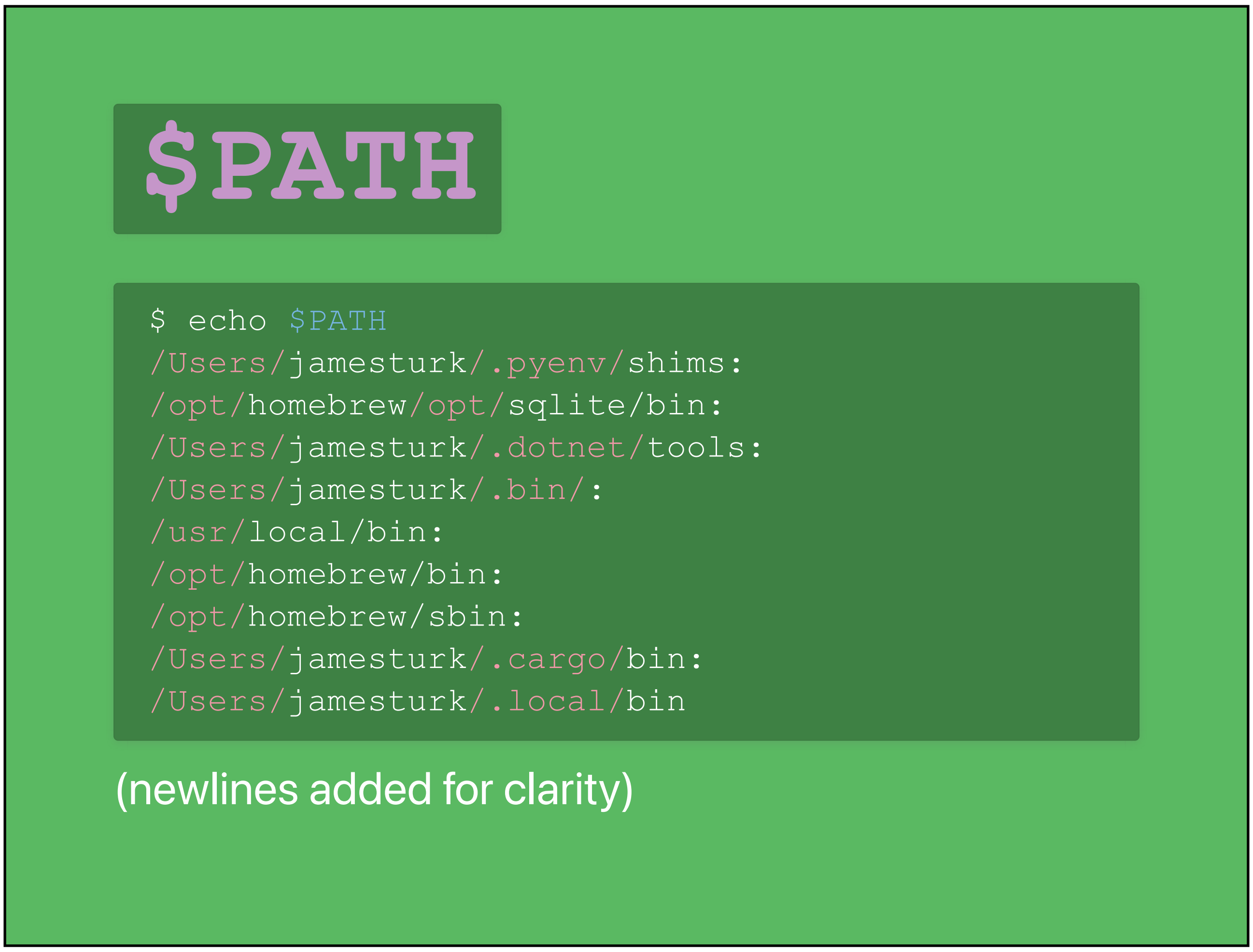
Executable Files

```
$ ls
file.sh
$ chmod a+x file.sh
$ ./file.sh
```

In the shell, we often use programs as commands. These in some ways are the shell equivalent of functions, ways to have repeatable behavior with optional parameters.

Commands are (mostly) executable files. Most commands you run in the terminal are actually executing some separate script or program somewhere on the computer.

(Sidestepping bash functions here for now, revisited in automation lecture.)



$PATH

```
$ echo $PATH
/Users/jamesturk/.pyenv/shims:
/opt/homebrew/opt/sqlite/bin:
/Users/jamesturk/.dotnet/tools:
/Users/jamesturk/.bin/:
/usr/local/bin:
/opt/homebrew/bin:
/opt/homebrew/sbin:
/Users/jamesturk/.cargo/bin:
/Users/jamesturk/.local/bin
```

(newlines added for clarity)

When you type a command in the terminal like: “python” or “git” what happens?

This variable is searched. (It is pretty crude actually, single string separated by colons)

So when you type `git`, the computer first tries to run the program `

/Users/jamesturk/.pyenv/shims/git

This doesn’t exist! (As you may have guessed, that directory has something to do with Python… more on that later.)

This is the first example of a fallback pattern that we're going to see in 3 different places this week.

`which`

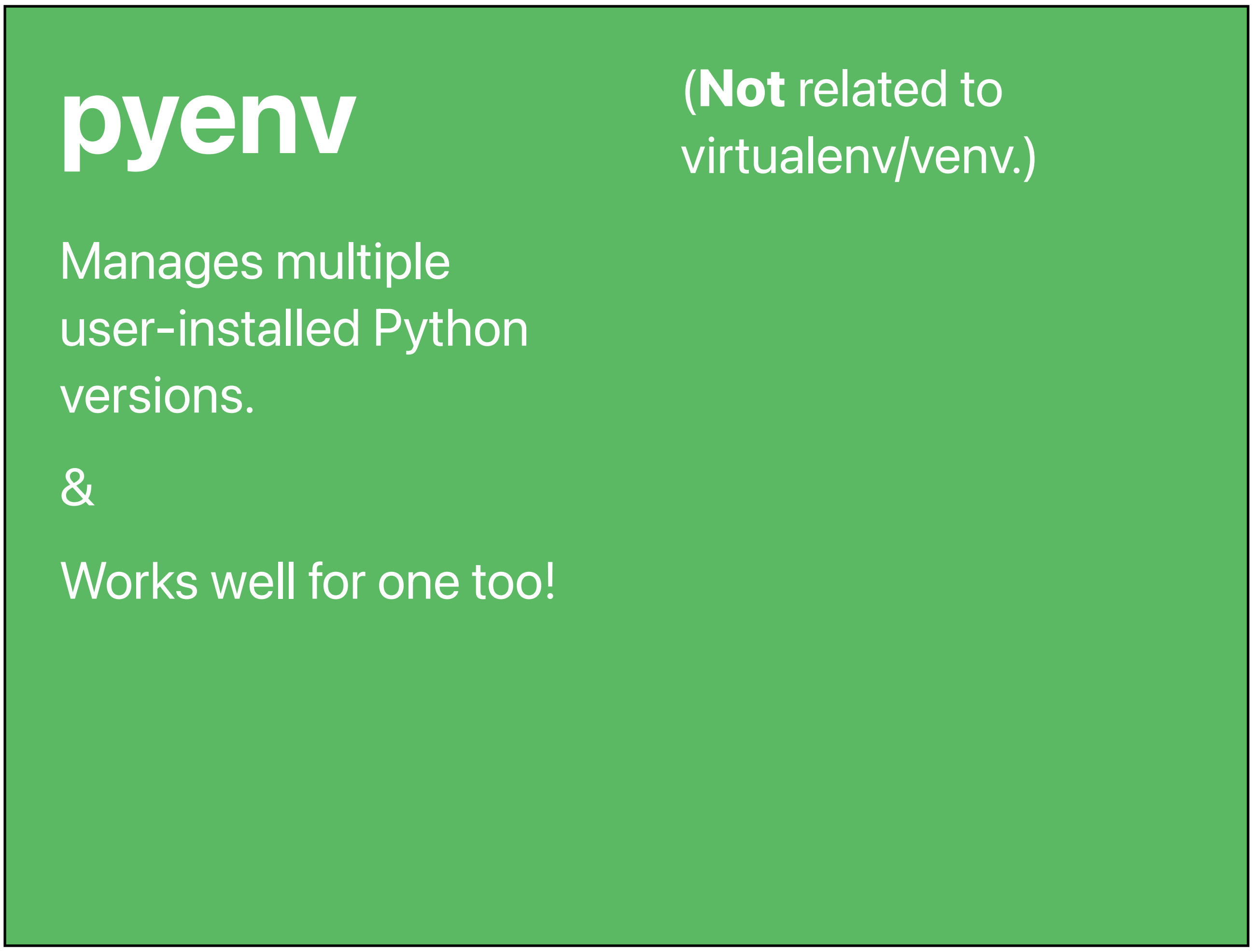
```
$ which python
/Users/jamesturk/.pyenv/shims/python
```

Path is searched in order until **first** match is found.

Multiple Pythons

Generally speaking it is something we're going to hope to avoid, but in many cases you do wind up with multiple version of Python installed.

This is a huge source of confusion, so I want to mention it here as a possibility, and now with `which` we know how to solve it.



pyenv

Manages multiple user-installed Python versions.

&

Works well for one too!

(**Not** related to virtualenv/venv.)

So I said this was typically something to be avoided, but in fact a lot of us do wind up actually wanting multiple versions. `pyenv` is a tool that lets you manage this.

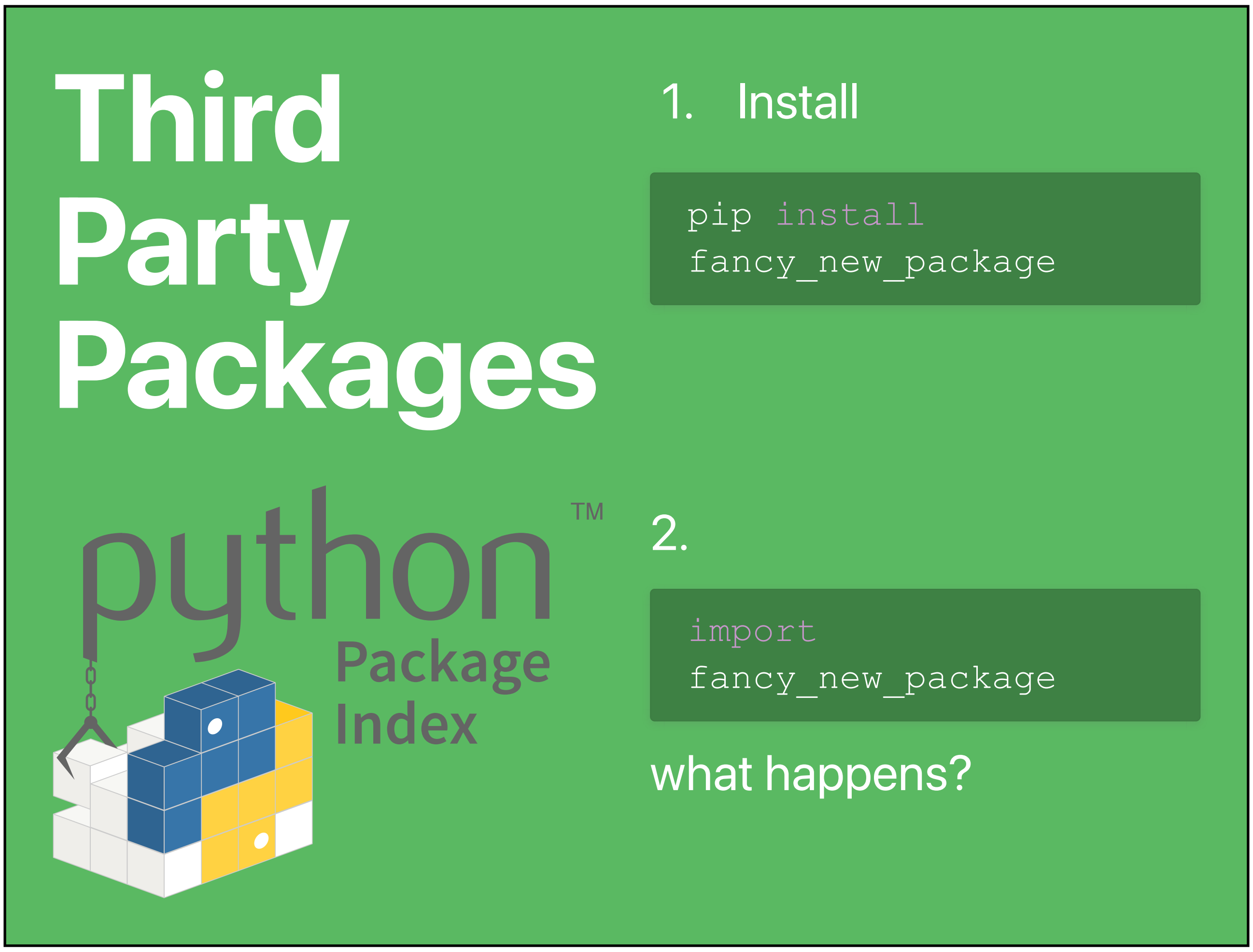
3. Python Packages

OK, time to get back to Python.

What do you do when you need some functionality that Python doesn't have built in?

One of the reasons Python is so popular is the ecosystem of third party packages. There are thousands of packages available for almost any task you can think of. You can find a list of packages on PyPI, the Python Package Index. (It is also possible to install packages from other sources, but this is the default & has hundreds of packages you might use.)

While packages like `math` and `csv` are built in to Python, other packages need to be installed. Some packages you've already used like `pytest` and `pandas` are installed on the CS Linux machines already. You may have found that if you tried to run code with those on your local machine it didn't work.



Third Party Packages

python™ Package Index

1. Install

```
pip install
fancy_new_package
```

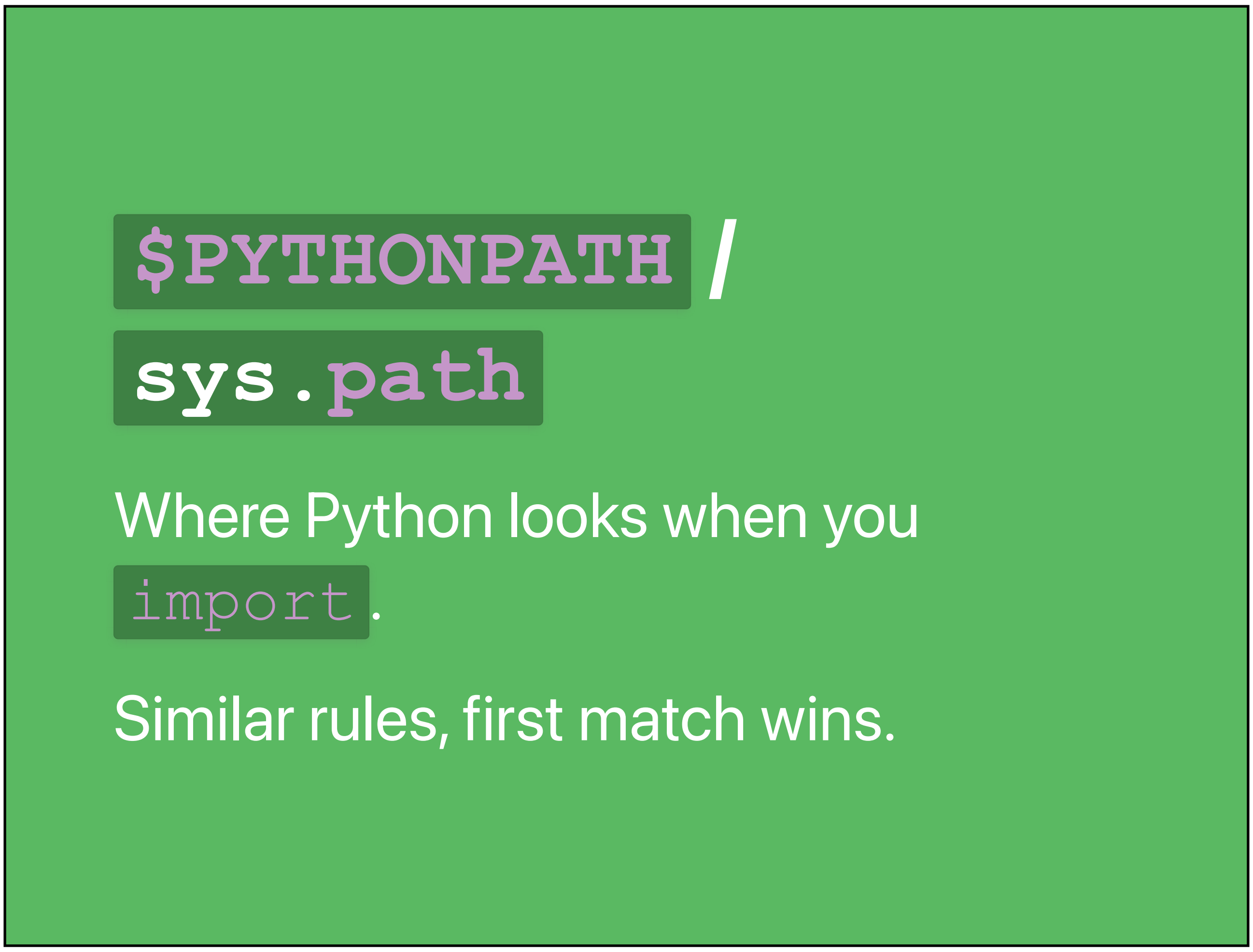
2.

```
import
fancy_new_package
```

what happens?

We'll come back to this.

How does import find what we just installed?



`$PYTHONPATH` / `sys.path`

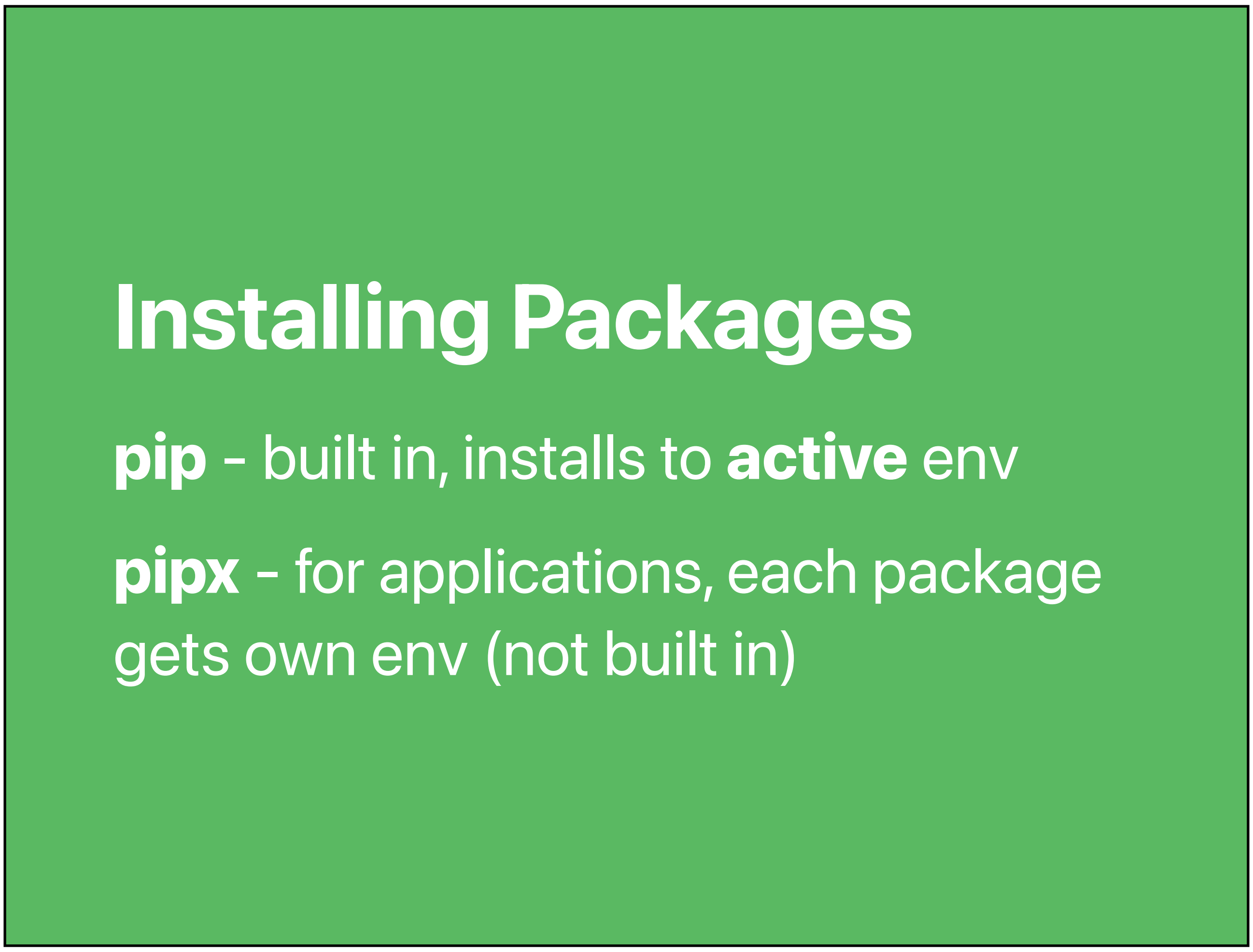
Where Python looks when you `import`.

Similar rules, first match wins.

Second instance of this pattern.

Third Party Packages

```
.../lib/3.11/site-packages/django
.../lib/3.11/site-packages/click
.../lib/3.11/site-packages/httpx
```



Installing Packages

pip - built in, installs to **active** env

pipx - for applications, each package gets own env (not built in)

It's possible to install packages from PyPI using a tool called `pip`. `pip` is a package manager that is installed with Python. You can install packages using `pip install package_name`. For example, to install `pytest` you would run `pip install pytest`.

Doing things this way installs packages globally. This can lead to issues when different programs rely on conflicting versions of packages. For this reason, it's best to install packages in a virtual environment.

`pipx` is a third party tool you can install that will ensure Python *applications* wind up with isolate environments.

Why does `pipx` not make sense for libraries then?

4. Virtual Environments

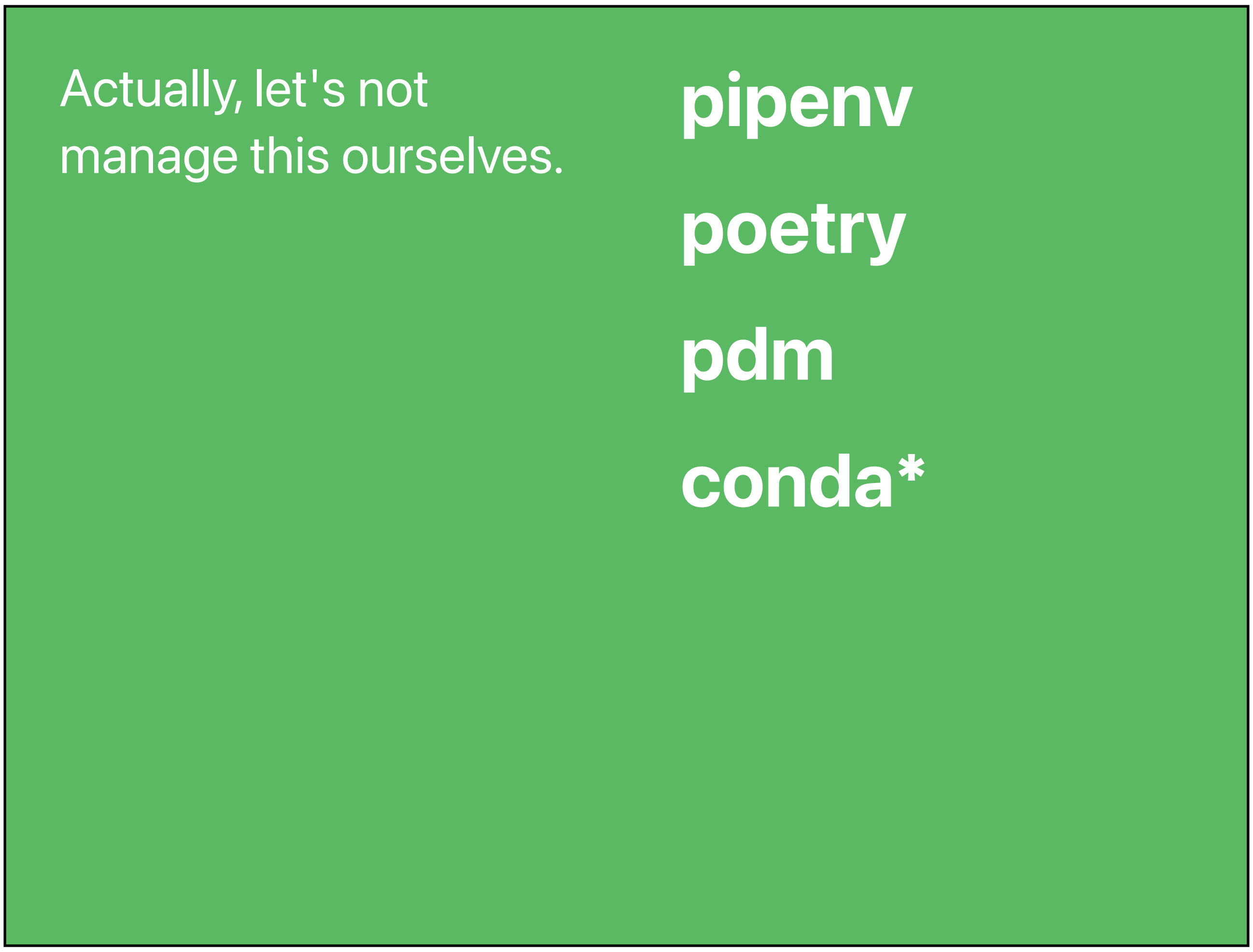
A virtual environment is a self-contained Python installation that contains its own set of packages. This allows you to have different versions of packages installed in different environments. This is especially useful when you are working on multiple projects that have conflicting requirements.

venv

a python interpreter + directory for packages

`venv` demo

- `python -m venv <dirname>` - create environment in a directory
- `<dirname>/bin/activate` - activate environment in a given session
- `rm <dirname>` - remove the environment
- `pip install -r requirements.txt` - install list of packages from a file



Actually, let's not manage this ourselves.

pipenv

poetry

pdm

conda*

This is widely recognized as the worst part of Python.

We can rely on tools to handle the worst parts, but each of them comes with a bit more of a learning curve. We're going to use `poetry`, which is a good balance of functional, popular, and fast.

(Why these 3 things?)

poetry, pdm, pipenv

Manually managing virtual environments can get tedious, if you plan to distribute your software at all you need to keep a list of dependencies around & make sure they are compatible with your code & as up to date as possible (for security fixes, etc.).

In the last few years, a new generation of tools have risen in popularity.

`poetry`, `pdm`, `pipenv` all essentially do the same thing, maintaining a list of dependencies for you as well as tools to build the virtualenv & run commands within it.

We'll be using `poetry` for this class, but the principles are virtually identical in `pdm` or `pipenv`.

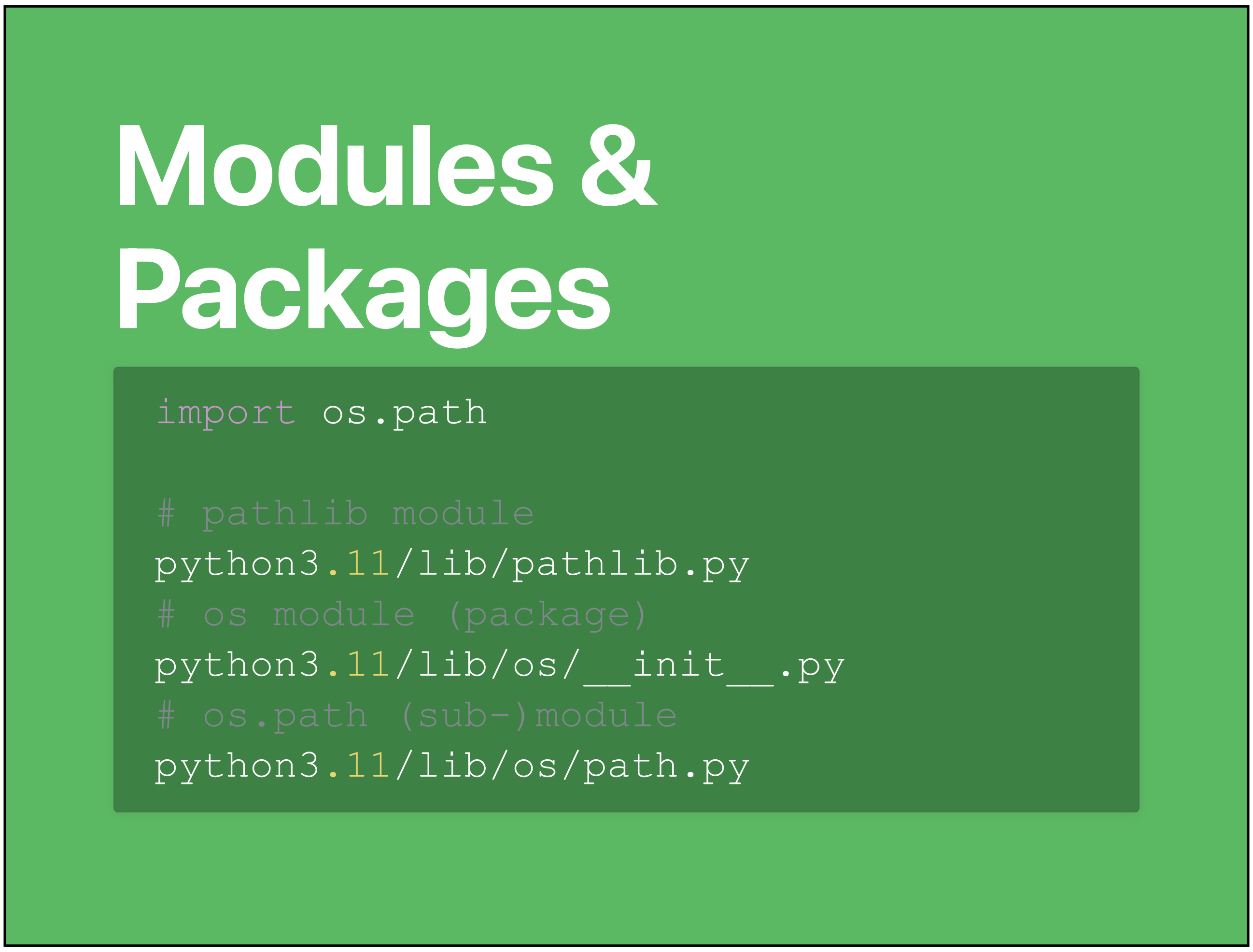
`poetry` demo

poetry cheatsheet

- `poetry install` - Install poetry packages from `pyproject.toml`
- `poetry add <pkgname>` - Add package to poetry environment (will update `pyproject.toml` and `poetry.lock`)
- `poetry remove <pkgname>` - Remove package from poetry environment.
- `poetry shell` - Same as activating virtualenv.
- `poetry run` - Run a command inside of virtualenv without "activating" it.
- `poetry init` - Create a new poetry project. (Not needed when a pyproject.toml already exists)

5. Python Applications

So that was a fairly long detour to get to our "primary" topic, Python applications. But with that understanding of how things work you are better suited to debug issues instead of just wondering why a command you know you installed is not found.



Modules & Packages

```
import os.path

# pathlib module
python3.11/lib/pathlib.py
# os module (package)
python3.11/lib/os/__init__.py
# os.path (sub-)module
python3.11/lib/os/path.py
```

(examples, actual Python structure may differ)

When you write Python, code lives in .py files.

These files are called modules. Modules are the basic unit of organization in Python. Modules can be imported into other modules or be run as a script.

Most non-trivial programs are made up of multiple files/modules.

- Code reuse: common functions can be shared between projects (e.g. `import math` or `import pathlib`).
- Namespace partitioning: Sometimes you have two functions with the same name, but they do different things. You can import them from different modules.

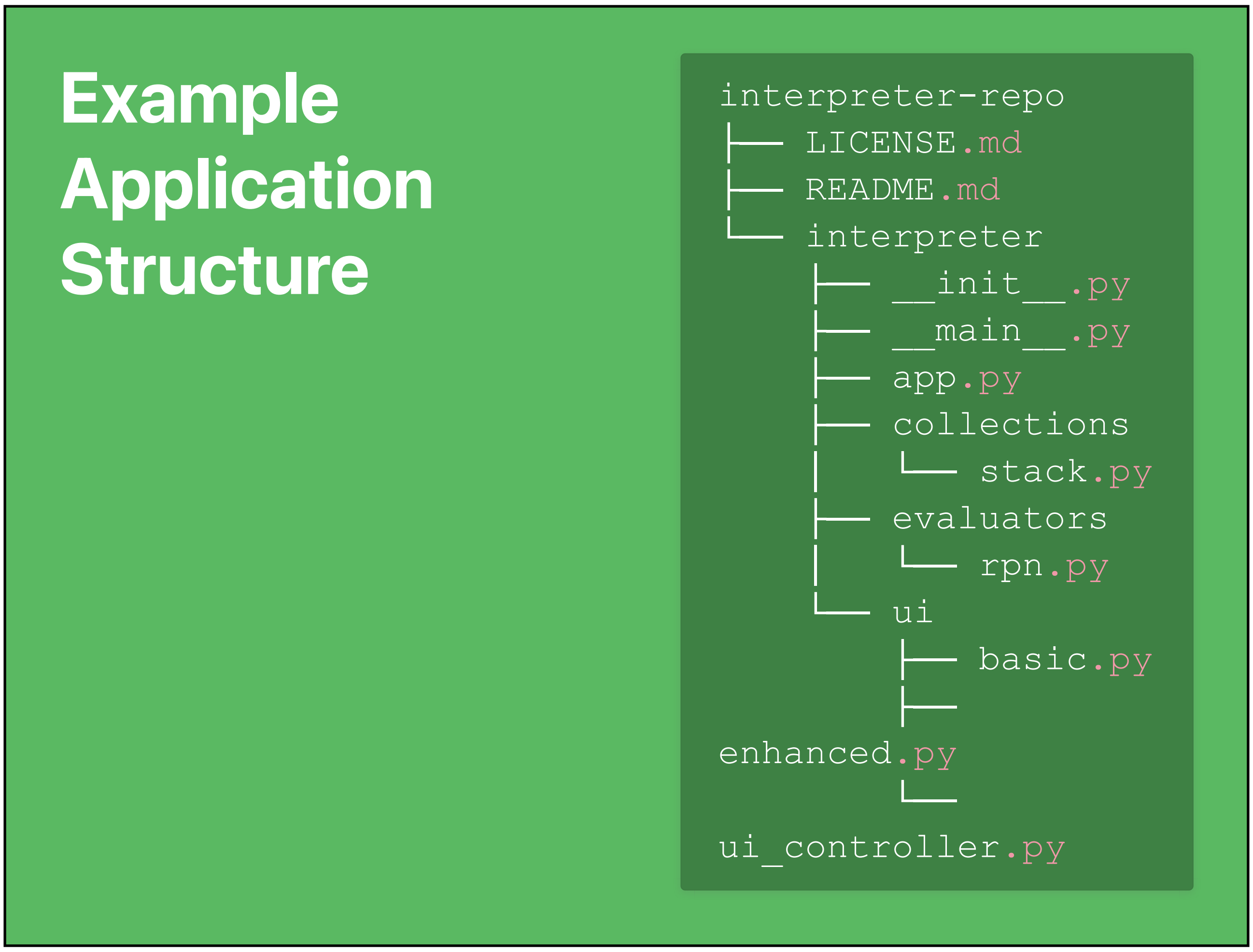
  (e.g. `import json` and `import yaml` both have a `load` function, but they do different things.)

Multiple related modules form a package. Packages are created by creating a directory with an `__init__.py` file in it. The `__init__.py` file can be empty, but it is required to make the directory a package.

You typically group modules based on shared purpose. A hypothetical application might have the following:

- `ui` package: contains modules related to the user interface
- `models` package: contains modules related to the data model
- `utils` package: contains modules with common utility functions (often a sort of catch-all)

These are completely up to you, the only requirement is that there is a top-level package directory with an `__init__.py` file. That file can be empty, but needs to be present to indicate the directory is a python package.



Example Application Structure

```
interpreter-repo
├── LICENSE.md
├── README.md
└── interpreter
    ├── __init__.py
    ├── __main__.py
    ├── app.py
    ├── collections
    │   └── stack.py
    ├── evaluators
    │   └── rpn.py
    └── ui
        ├── basic.py
        ├──
enhanced.py
        └──
ui_controller.py
```

To help us get started thinking about the differences between packages and modules, here's the structure of a simple arithmetic interpreter (located in `m1/interpreter-repo`):

Notice in the root directory (`interpreter-repo`) there is a single folder called `interpreter`. This directory (i.e., package) is known as the *top-level* package, and all the packages underneath are known as *subpackages*. The subpackages are `collections`, `evaluators`, `ui`. The top-level package also contains a special file in `__main__.py`. This file is run when we execute our top-level package directly via `python3 -m interpreter`. We'll talk about what goes in that `__main__.py` later.

import review

```
# two main import forms
import module_name
from module_name import
module_attr

# aliased
import module_name as alias
from module_name import
module_attr as alias
```

Now's a good time to review the different ways you can import modules and packages.

1) `import module_name`

Imports the module and makes it available in the current namespace. You can access the module's functions by prefixing them with the module name. For example, for the module `math` with a function called `sin`, you can access it by calling `math.sin()`.

2) `from module_name import module_attr`

Imports a specific attribute from a module and makes it available in the current namespace. For example, `from math import sin` will import the `sin` function from the `math` module and make it available in the current namespace. You can then call it directly by calling `sin()`.

3) `import module_name as alias` or `from module_name import module_attr as alias`

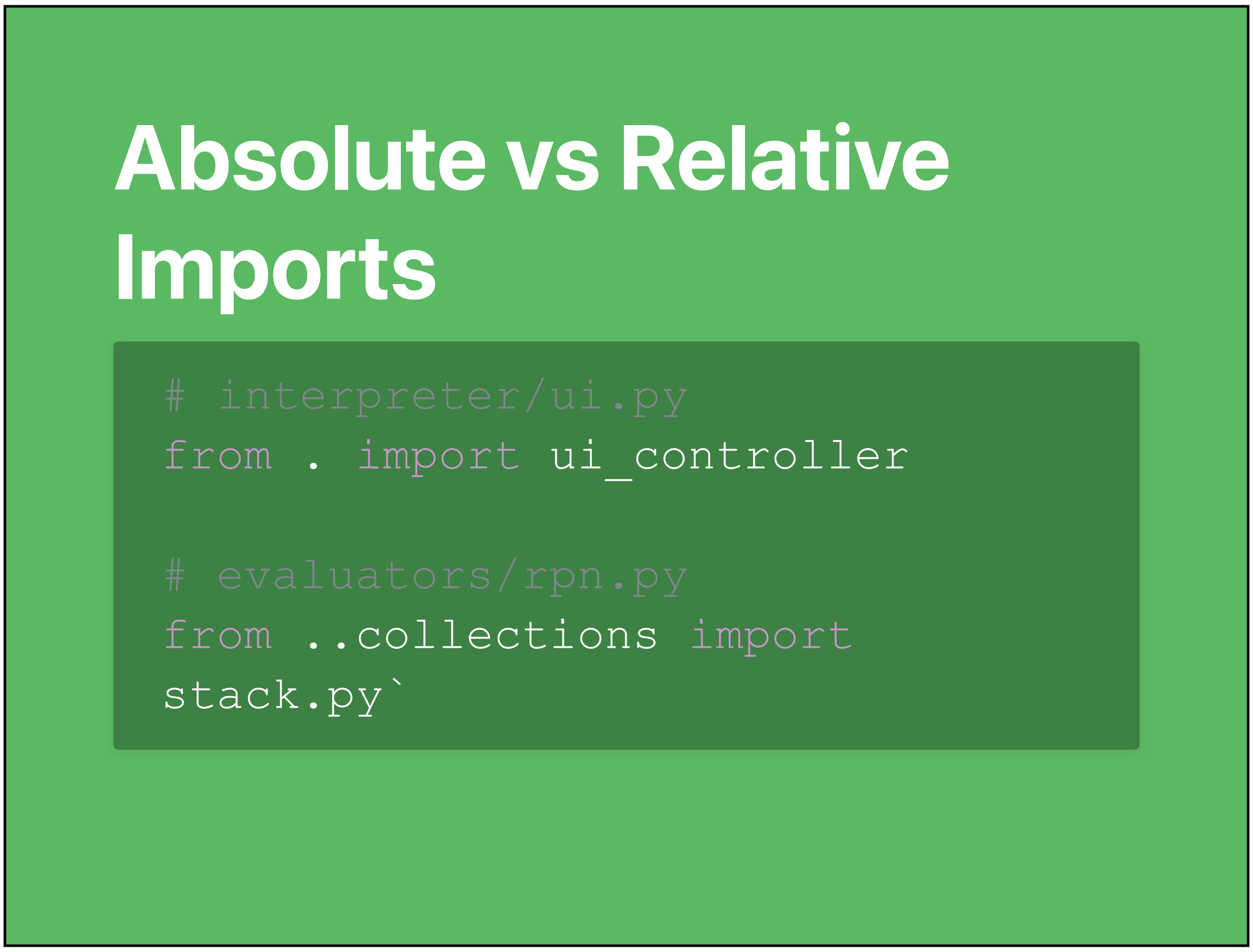
Imports a module or attribute and gives it an alias. For example, `import pandas as pd` will import the `pandas` module and make it available as `pd`. You can then access the `DataFrame` class as `pd.DataFrame`. This is commonly used in data science libraries (`import numpy as np`, `import pandas as pd`, etc) but overuse can make code harder to read. It's best to use aliases sparingly.

4) `from module import *`

Makes contents of module available in the current namespace. This is considered bad practice and should not be used. It can lead to namespace collisions and make it hard to tell where functions are coming from.

```
from math import *
```

No.



Absolute vs Relative Imports

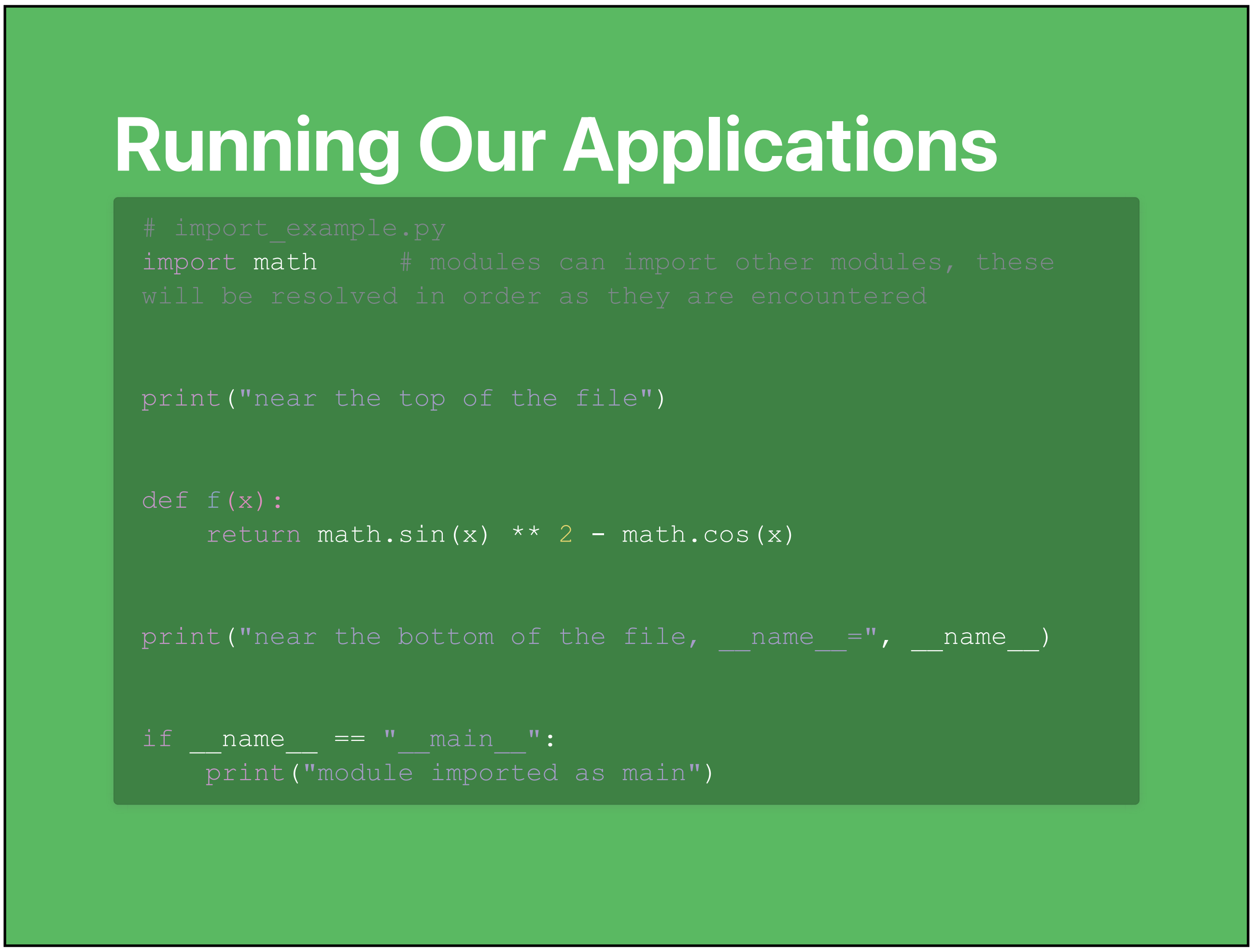
```
# interpreter/ui.py
from . import ui_controller

# evaluators/rpn.py
from ..collections import
stack.py`
```

Python has two types of imports: absolute and relative. Absolute imports are imports that start from the top-level package. Relative imports are imports that start from the current module. Relative imports are denoted by a `.`.

In our above application, from outside the package you might import the `ui` package via `import interpreter.ui`. This is an absolute import. If you are inside the `ui` package, you might import the `ui_controller` module via `from . import ui_controller`. This is a relative import.

Relative imports allow you to rename/refactor packages with fewer updates to your imports.

For more details you may want to read <u>Absolute vs. Relative Imports</u>.



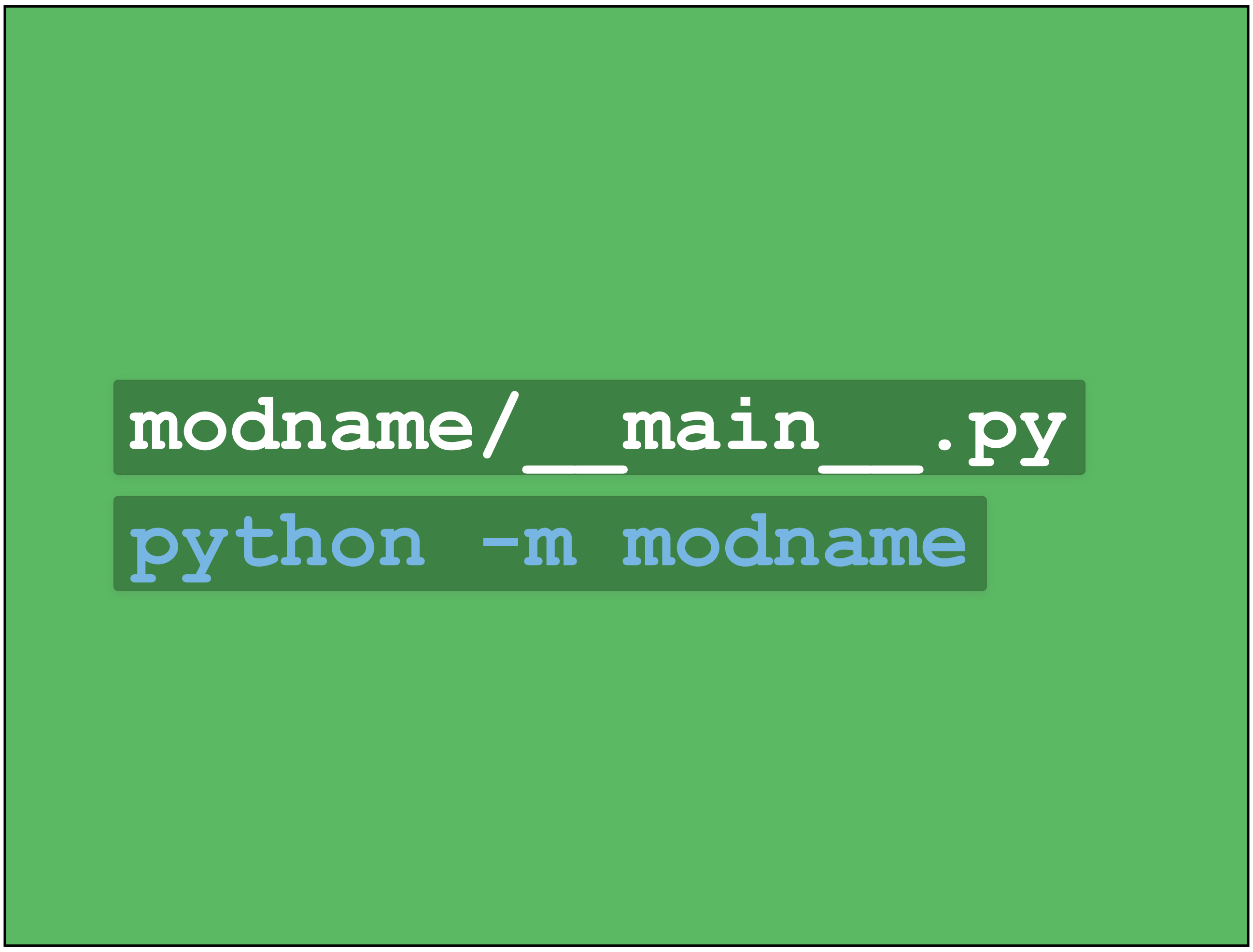
Running Our Applications

```
# import_example.py
import math      # modules can import other modules, these
will be resolved in order as they are encountered


print("near the top of the file")


def f(x):
    return math.sin(x) ** 2 - math.cos(x)


print("near the bottom of the file, __name__=", __name__)


if __name__ == "__main__":
    print("module imported as main")
```

Running `python3 import_example.py` will print `Running module directly`. Importing the module will not print anything, but will allow access to the `func` function.

Demo 1: executing the file

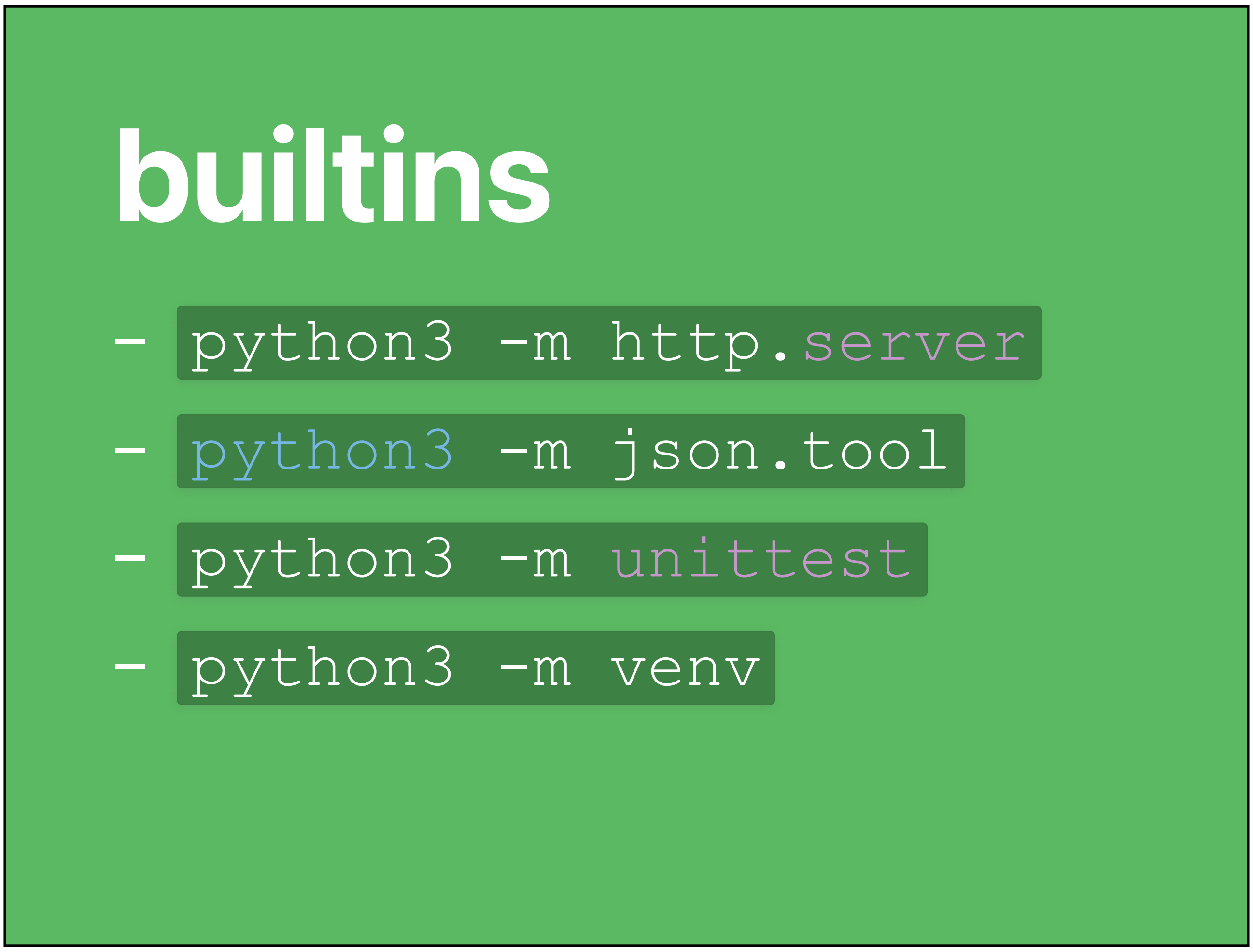
Demo 2: importing the file



```
modname/__main__.py
python -m modname
```

The `__main__.py` file is a special file that is run when you execute a package directly. For example, if you have a package called `foo`, you can run it by executing `python -m foo`. The `__main__.py` file is the entry point for the package.

An application like the one above, it is common to make the top-level package executable. To execute a package you can run `python -m package_name`. This will execute the `__main__.py` file in the package.



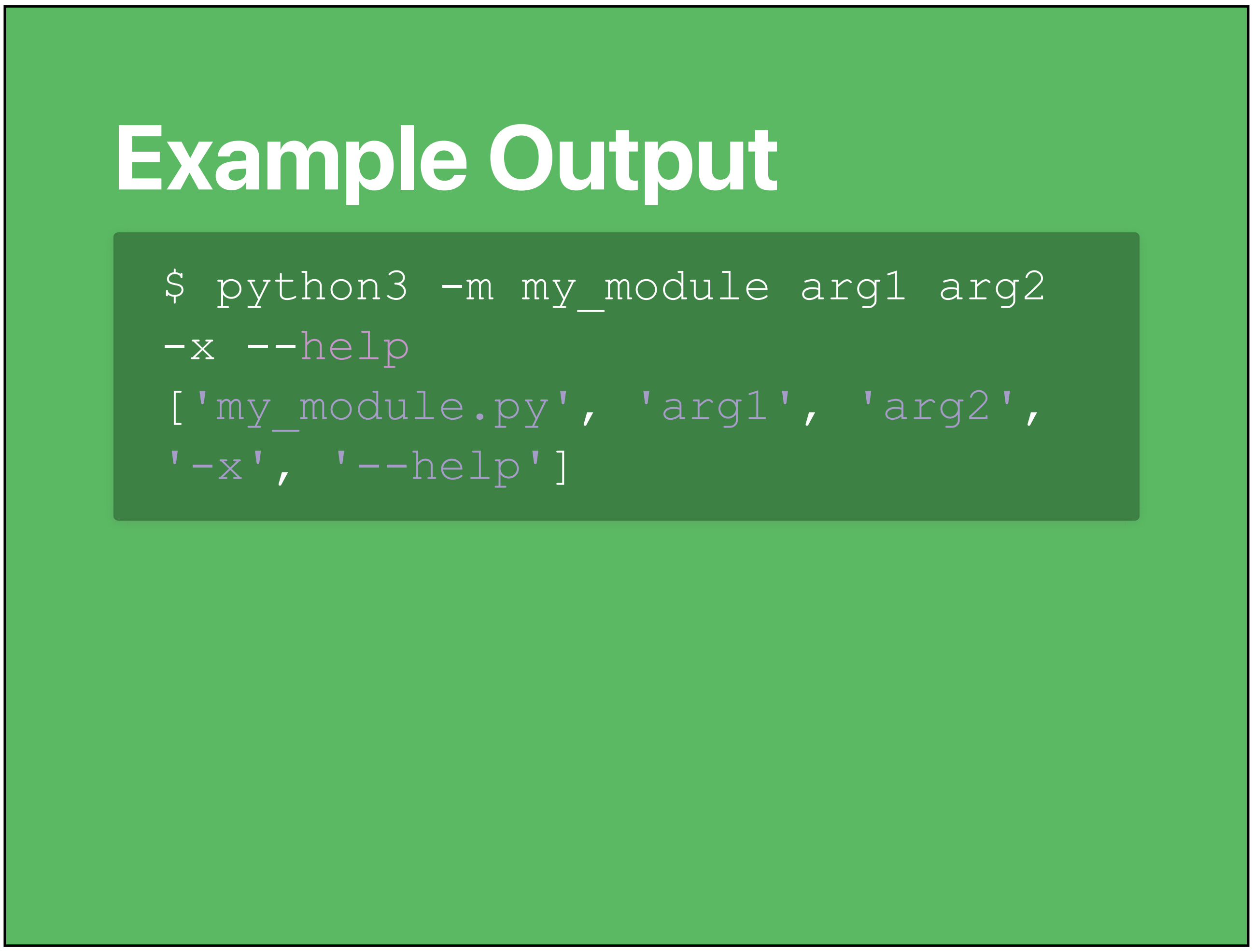
builtins

- `python3 -m http.server`
- `python3 -m json.tool`
- `python3 -m unittest`
- `python3 -m venv`

Some built in modules have an executable package.

Command Line Arguments

```python
# my_module.py
import sys

if __name__ == "__main__":
    print(sys.argv)
```



Example Output

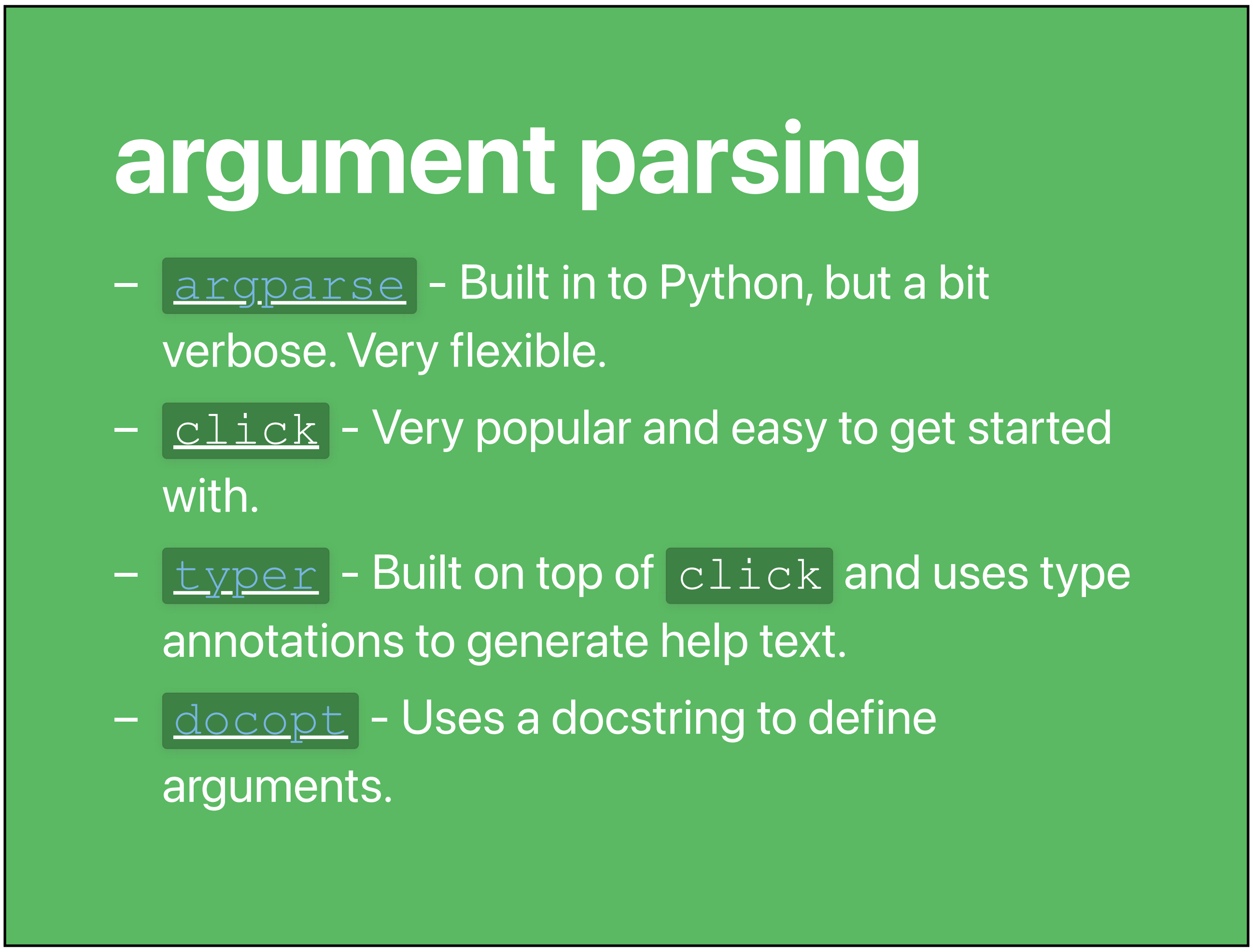
```
$ python3 -m my_module arg1 arg2
-x --help
['my_module.py', 'arg1', 'arg2',
'-x', '--help']
```

Whichever way you run a Python module, you can pass command line arguments to it. These arguments wind up in a special list called `sys.argv`. The first element is the name of the module. The second element is the first command line argument, and so on.

These arguments are available in the `sys.argv` list. The first argument is the name of the module. The second argument is the first command line argument, and so on.

If you've used the command line much, you're probably familiar with programs that take all sorts of flags and arguments.

There are several modules that help with turning command line arguments into a more structured format.



argument parsing

- `argparse` - Built in to Python, but a bit verbose. Very flexible.
- `click` - Very popular and easy to get started with.
- `typer` - Built on top of `click` and uses type annotations to generate help text.
- `docopt` - Uses a docstring to define arguments.

%% Conclusion

With that we've seen how command line apps work and how our own apps can receive arguments from the command line. We'll be building complete applications in all of our PAs this year and for our final project.

We'll revisit the command line a bit more in a few weeks, and next class we'll get back into writing Python with a discussion of inheritance and Python's data model.